

# **FORTH Utilities - Assembler and Discompiler for ATOM FORTH**

## **Part 1 - The FORTH Assembler**

### **1 Introduction**

When writing long applications there will be occasions when even the high speed of FORTH is not sufficient and unacceptably long execution times will occur. In addition certain applications, such as interrupt handlers, can not be written entirely in a high-level language. In such circumstances it is necessary to resort to machine code.

Short machine-code routines can be hand-assembled, placing the appropriate bytes of code directly into the dictionary (see "FORTH Theory and Practice", Section 5.4). In a small system this has the advantage of requiring no additional software aids, but has a number of significant drawbacks. It can, for example, be a very tedious process for all but the shortest routines, and the resulting source code is virtually unreadable.

The addition of an assembler removes these difficulties at the expense of increasing the software overheads. A FORTH assembler, however, occupies only about 1.5 K and greatly simplifies the task of producing error-free code. In many cases the application can be developed in high-level FORTH and the time-critical sections subsequently replaced by the machine-code equivalents. Such a development will take very little more time than an entirely high-level approach.

The use of a FORTH assembler may appear somewhat strange to anyone accustomed to a 'conventional' assembler. The branch instructions are never used explicitly, the assembly mnemonics and their operands are written in the 'wrong' order, and labels are only rarely required.

It is, however, an extremely sophisticated single-pass assembler with comprehensive error checks. Furthermore, all the power of FORTH itself is available throughout the assembly process. The speed, power and simplicity far outweigh the need to become accustomed to the unusual approach.

## 2 A Simple Example

We can illustrate some of the features of the FORTH assembler by looking at the creation of a simple definition.

Before trying this or any other example the assembler must be loaded. It is provided in ten screens of source code, starting at screen 24, and is loaded in the normal way by typing:

```
24 LOAD
```

The example we shall look at is DROP whose action should not need explaining. The code for this definition could be written in conventional assembly language as:

```
4C 20 29          JMP POP
```

(Remember that POP is the entry point to machine code which removes the top item from the stack and then executes NEXT to move on to the following word.)

The FORTH assembler allows the word with this code to be created by:

```
CODE DROP  POP JMP,  END-CODE
```

The two words CODE and END-CODE are used to start and end a code definition, rather like <:> and <;> are used for a colon-definition.

CODE generates the name header and starts some of the error-checking procedures. Unlike <:>, however, it changes the CONTEXT vocabulary to ASSEMBLER rather than to the CURRENT vocabulary.

END-CODE, in addition to performing a number of error checks, restores the original CONTEXT vocabulary.

Two points should be noted here which may help with the use of the assembler. Firstly, the assembler mnemonic (i.e. JMP,) ends with a comma. All assembler mnemonics have a terminating comma which serves three purposes:

- 1 It marks the end of a group of words which would be a single line of conventional assembly code.
- 2 The FORTH word <,> compiles values into the dictionary, and so, by analogy, the comma indicates the point at which bytes of code are actually compiled.

- 3 It ensures that there is no confusion between assembly mnemonics and hexadecimal numbers (e.g. ADC) or other assembler words (e.g. SEC).

Secondly, the operand (i.e. POP) is placed before the mnemonic rather than afterwards as in the conventional assembler. This is always the case whether the operand is a literal numeric value or, as in the above example, a symbolic one. The operand, in either case, causes its value to be placed on the stack ready to be compiled together with the opcode by the execution of the assembly mnemonic.

This illustrates the main difference between code assembly and the generation of a colon-definition. During the assembly process the source text is executed rather than compiled. Assembly is therefore simply a normal interpretation of the text with the ASSEMBLER vocabulary as CONTEXT (i.e. searched first). This means that the whole of FORTH is also available during the assembly for modifying or manipulating the stack contents, making the process extremely flexible.

In effect, each assembly mnemonic is a mini-compiler which executes to compile its own opcode and operand (if any). During the normal assembly process it is execution time for the assembler words but compile time for the definition being generated.

The assembler words may also be compiled into a colon-definition if desired. This will defer their action until this colon-definition is executed and is the basis of macro assembly (see later).

### 3 Machine-Code Labels

All machine code must end with a jump to code which will eventually execute NEXT. The terminating routines which are provided are:

|        |                                                   |
|--------|---------------------------------------------------|
| NEXT   | proceed to next word                              |
| PUT    | replace top stack item                            |
| PUSH   | push from accumulator and hardware stack to stack |
| PUSH0A | push zero and accumulator to stack                |

POP        drop top stack item  
POPTWO    drop top two stack items

These are described more fully in section 5.4 of "FORTH Theory and Practice". In the assembler they are defined as constants which return the appropriate address.

Another defined constant is SETUP. This returns the address of a subroutine which transfers up to four values from the stack to the scratchpad area. On entry the accumulator should contain the number of items to be transferred. For example:

```
... 2 # LDA, SETUP JSR, ...
```

will transfer two items.

#### 4 Registers

FORTH uses both the processor registers and a number of special registers located in page zero. The assembler contains, again as defined constants, the addresses of all the zero page registers used. These are:

|       |                          |
|-------|--------------------------|
| W     | codefield pointer        |
| IP    | interpretive pointer     |
| UP    | user area pointer        |
| XSAVE | temporary for X-register |
| N     | scratchpad               |

The contents of the first three of these registers are fundamental to the operation of FORTH and should be used with great care. If their contents are changed inadvertently the system will certainly 'go away'! Their functions are fully described in Section 5 of "FORTH Theory and Practice". Examples of the use of these registers are given later.

All the 6502 processor registers are used by FORTH. They may also be used in a machine code definition, provided that certain conventions are observed. Whenever a machine code definition is executed, it is entered directly from the routine NEXT. The processor registers will therefore always be subject to the same conditions, which are as follows:

- 1 The contents of the accumulator are undefined. The accumulator may be used freely in any machine-code definition.

- 2 The Y-register contains zero and may be used without restriction.
- 3 The X-register points to the low byte of the number on the top of the computation stack. It may be used provided that its contents are first saved (in XSAVE) and then restored at the end of the routine.
- 4 The hardware stack pointer points to the first free byte below the bottom of the return stack (this is normal for the 6502, whose stack grows downwards in memory). Its contents should not normally be changed across a code definition, otherwise FORTH return addresses may be lost.
- 5 The contents of the processor status register, with the exception of the decimal flag, are undefined and may be used freely. The decimal flag is clear, so that the processor is in binary mode, and must be returned in that state.

## 5 Opcode Mnemonics

The assembly mnemonics are divided into two main groups according to their addressing modes. The first group contains the one-byte codes with only a single (implied) addressing mode. The second group contains those that have a number of different addressing modes. All the mnemonics used are the standard 6502 mnemonics with an additional terminating comma.

### 5.1 Single-Mode Mnemonics

The single-mode mnemonics are as follows:

|      |      |      |      |      |
|------|------|------|------|------|
| BRK, | CLC, | CLD, | CLI, | CLV, |
| DEX, | DEY, | INX, | INY, | NOP, |
| PHA, | PHP, | PLA, | PLP, | RTI, |
| RTS, | SEC, | SED, | SEI, | TAX, |
| TAY, | TSX, | TXA, | TXS, | TYA, |

When any of these is executed it simply compiles the corresponding opcode into the dictionary.

The following example will execute to push a zero to the computation stack:

CODE ZERO TYA, PHA, PUSH JMP, END-CODE

It uses the fact that the Y-register always contains zero on entry to a code routine. On entering PUSH, therefore, the bottom byte of the hardware stack and the accumulator will both contain zero and it is these bytes (accumulator first) which are transferred to the stack by PUSH.

## 5.2 Multi-Mode Mnemonics

The multi-mode mnemonics are as follows:

|      |      |      |      |      |
|------|------|------|------|------|
| ADC, | AND, | ASL, | BIT, | CMP, |
| CPX, | CPY, | DEC, | EOR, | INC, |
| JMP, | JSR, | LDA, | LDX, | LDY, |
| LSR, | ORA, | ROL, | ROR, | SBC, |
| STA, | STX, | STY, |      |      |

Each of these normally needs an operand which must previously have been placed on the stack. If no address mode is given the operand is assumed to be an address. In this case absolute (16-bit) or zero page ( 8-bit) modes are chosen automatically, depending on the magnitude of the address and on which modes are legal for the particular opcode.

When any of these mnemonics is executed it compiles the appropriate opcode and operand into the dictionary.

## 5.3 Addressing Modes

The symbols in the following table are used to specify the addressing mode to be used:

| Symbol | Mode               | Expected Operand        |
|--------|--------------------|-------------------------|
| None   | memory             | zero page or absolute   |
| .A     | accumulator        | none                    |
| #      | immediate          | eight bit literal value |
| ,X     | indexed X          | zero page or absolute   |
| ,Y     | indexed Y          | zero page or absolute   |
| X)     | indexed indirect X | zero page               |
| )Y     | indirect indexed Y | zero page               |

)     indirect                     absolute

(.A is used to make a distinction from hexadecimal A)

The way in which the addressing modes are used is illustrated below. The FORTH version is compared with the corresponding form for a conventional assembler.

| FORTH        | Conventional assembler |
|--------------|------------------------|
| ADDR JSR,    | JSR ADDR               |
| .A ASL,      | ASL A                  |
| 4 # CMP,     | CMP #4                 |
| ADDR ,X LDA, | LDA ADDR,X             |
| ADDR ,X STA, | STA ADDR,Y             |
| ADDR X) ADC, | ADC (ADDR,X)           |
| ADDR )Y LDA, | LDA (ADDR),Y           |
| ADDR ) JMP,  | JMP (ADDR)             |

The following two examples make use of the OSWRCH routine, at address #FFF4, of the Acorn ATOM operating system. This displays the contents of the accumulator as an ASCII character. Since all the processor registers are preserved by this routine, no special precautions need to be taken to restore them.

The word CHAR displays the content of the byte at address #80 as an ASCII character.

HEX

```
CODE CHAR ( ... )
  80 LDA, FFF4 JSR, NEXT JMP,
END-CODE
```

DECIMAL

The following sequence typed at the keyboard will then display an 'A' on the VDU (note that we are back in decimal mode).

65 128 C! CHAR

If the routine is modified to use the 'indirect indexed Y' mode it will display the character whose address is given by the two bytes starting at address #80:

HEX

```
CODE (CHAR) ( ... )
  80 )Y LDA, FFF4 JSR, NEXT JMP,
END-CODE
```

## DECIMAL

This routine again makes use of the fact that the Y-register contains zero on entry.

Typing the sequence below will display a 'B' (the first character in the name header of BASE) on the display:

```
' BASE NFA 1+ 128 ! (CHAR)
```

## 6 Accessing the stacks

Most machine-code routines will need to access the computation stack, the return stack or both. A separate technique is needed for each stack.

Since both stacks grow towards low memory, it is inappropriate in the context of machine code, to call the most-accessable number the 'top' item. In a FORTH assembler, it is usually referred to as the bottom stack value. In the text of this manual, the word 'top' will continue to be used but will be placed in quotes as a reminder.

### 6.1 The Computation Stack

This stack is in page zero and is usually addressed in zero page ,X mode with the X-register as the stack pointer. The corresponding addresses are illustrated in the following diagram:

| Stack             | addressing |
|-------------------|------------|
| second, high byte | 3 ,X       |
| second, low byte  | 2 ,X       |
| bottom, high byte | 1 ,X       |
| bottom, low byte  | 0 ,X       |

These stack items are accessed so frequently that the words BOT and SEC are provided. Thus, for example:

```
    BOT LDA, is equivalent to    0 ,X LDA,
SEC 1+ STA, is equivalent to    3 ,X STA,
```



As an example we can consider the following definition of 2\* which performs a fast multiply-by-two. Though it is one byte longer, it is even faster than the one provided in ATOM FORTH!

```
CODE 2*  ( n1 ... n2 )  
  BOT ASL, BOT 1+ ROL, NEXT JMP,  
END-CODE
```

## 6.2 The Return Stack

The return stack is located in the 6502 hardware stack, in page one. Although the hardware stack pointer points one byte below the bottom of the stack, the following sequence will remove the bottom item (low byte first):

```
PLA, PLA,
```

If it is necessary to manipulate return stack values, the hardware stack pointer can be transferred to the X-register. The X-register normally contains the computation stack pointer so its contents must first be saved and later restored using the temporary storage location XSAVE.

The special address modifier RP) is provided to access the bottom byte of the return stack. The instruction

```
RP) LDA, (equivalent to 101 ,X LDA,)
```

will load the accumulator with the bottom byte of the return stack. The following code will, for example, load the accumulator with the low byte of the second return stack item (i.e. the third byte from the bottom of the hardware stack):

```
XSAVE STX, TSX,  
RP) 2+ LDA,  
XSAVE LDX,
```

The code has been divided into three lines. The first and third lines will normally be used to enclose any reference, or group of references, to the return stack.

## 7 Conditional Structures

The following conditional branching structures are provided (note the additional terminating comma for the assembler versions):

```
IF, ... ELSE, ... THEN,  
BEGIN, ... AGAIN,  
BEGIN, ... UNTIL,  
BEGIN, ... WHILE, ... REPEAT,
```

They are used in a similar way to the high-level equivalents but with one important difference. Whereas the high-level structures test the 'top' stack item, the assembler versions test various bits in the status register. The tests provided are as follows:

| Test   | Meaning                 | True for |
|--------|-------------------------|----------|
| CS     | carry set               | C=1      |
| CS NOT | carry clear             | C=0      |
| VS     | overflow                | V=1      |
| VS NOT | overflow clear          | V=0      |
| 0<     | byte less than zero     | N=1      |
| 0< NOT | byte not less than zero | N=0      |
| 0=     | byte equals zero        | Z=1      |
| 0= NOT | byte Not equal to zero  | Z=0      |

The following examples illustrate some uses of the conditional structures. The first gives a faster alternative to 1+ (which has a high-level definition in the nucleus):

```
CODE 1+ ( n1 ... n2 )  
  BOT INC,  
  0= IF, BOT 1+ INC, THEN,  
  NEXT JMP,  
END-CODE
```

The second example illustrates the use of IF, ... ELSE, ... THEN, and the nesting of conditionals. It increments or decrements the second stack item by one, depending on the sign of the 'top' stack number.

```
CODE ?INC/DEC ( n1\n2 ... n3 )  
  BOT 1+ LDA,  
  0< IF,      SEC LDA,  
              0= IF, SEC 1+ DEC, THEN,  
              SEC, DEC,  
  ELSE,      SEC, INC,
```

```

        0= IF, SEC 1+ INC, THEN,
    THEN,
    POP JMP,
END-CODE

```

The next two examples implement double-precision fetches and stores. They work in the same way as <@> and <!> except that the value occupies four bytes starting at addr instead of two.

```

CODE 2@ ( addr ... nd )
    1# LDA, SETUP JSR, 3 # LDY,
    BEGIN, N )Y LDA, DEX, BOT STA, DEY, 0= UNTIL,
    0= UNTIL,
    NEXT JMP,
END-CODE

```

```

CODE 2! ( nd\addr ... )
    1 # LDA, SETUP JSR,
    BEGIN, BOT LDA, N )Y STA, INX,
        INY, 4 # CPY,
    0= UNTIL,
    NEXT JMP,
END-CODE

```

## 8 Compiler Security and Unstructured Code

According to the rules of structured programming, loops and conditional branches may be nested, provided that each inner structure is completely enclosed within the outer one. On occasions, however, it may be necessary to produce unstructured machine code. To do this it is necessary to manipulate the address and security check digits left on the stack by the conditionals. A summary of these is given in the following table. The stack action of each word is shown in the usual notation. The symbol s represents a status check byte.

|         |    |                 |                     |     |
|---------|----|-----------------|---------------------|-----|
| BEGIN,  | (  | ...             | addrB\1             | )   |
| AGAIN,  | (  | addrB\1         | ...                 | )   |
| UNTIL,  | (  | addrB\1\s       | ...                 | )   |
| WHILE   | (  | addrB\1\s       | ... addrB\1\addrW\2 | )   |
| REPEAT, | (  | addrB\1\addrW\2 | ...                 | )   |
| IF,     | (  | s               | ... addrI\2         | )   |
| ELSE,   | (  | addrI\2         | ... addrE\2         | )   |
| THEN,   | (  | addrI\2         | ...                 | )   |
|         | or | (               | addrE\2             | ... |
|         |    |                 | )                   |     |

A common use is to assemble an unstructured branch out of the current definition. For example the sequence:

```
ADDR 1 0= UNTIL,
```

assembles the code for

```
BNE ADDR
```

Remember that UNTIL, (and the high-level version) cause a branch to be taken UNTIL the condition is met.

Great care is needed when manipulating the stack in all but the simplest cases and it is wise to work it out on paper beforehand.

## 9 ;CODE

In high-level definitions new data structures can be created with the aid of <BUILDS ... DOES> defining words. The action of a word created by such a defining word is governed by the words following DOES>.

Defining words with a machine code action may be created by use of the pair

```
CREATE ... ;CODE
```

The general principles are the same as in the use of <BUILDS and DOES>. The words following CREATE compile the parameters peculiar to the family member but all members execute the code which follows ;CODE. As well as starting the assembler section, ;CODE terminates the colon-definition and <;> is not used.

A couple of examples should help to make the action clear. The first allows the creation of double-precision variables.

```
: 2VARIABLE      ( ... addr )  
    CREATE      ( the name header )  
    , ,        ( compile the double number ).  
    ;CODE      ( push PFA to stack )  
        CLC,   W LDA,  2 # ADC,  PHA,  
        TYA,   W H ADC,  PUSH JMP,  
    END-CODE
```

The sequence:

```
1234567.  2VARIABLE  DOUBLE
```

will create a double precision variable, DOUBLE, whose initial value is 1234567. When DOUBLE is used it will execute the machine code following ;CODE in 2VARIABLE to leave on the stack the address of the first byte of the parameter field of DOUBLE.

A value may be fetched from or stored to DOUBLE by the use of 2@ or 2! which were defined earlier.

The code used by 2VARIABLE is identical to that of VARIABLE since both return the same address. Memory use can be minimised without any loss of execution time by the following alternative definition:

```
: 2VARIABLE  VARIABLE , ;
```

However, this is not a very good illustration of how ;CODE works !

As a second example we can look at a possible definition of 2CONSTANT to create double-precision constants. Since such a word must copy four bytes to the stack it cannot use the machine code of CONSTANT.

The following version is designed for clarity and to minimise memory usage rather than for maximum execution speed, though in fact it is still quite fast. It requires the prior machine code definition of 2@.

```
: 2CONSTANT  ( ... nd)
      CREATE  , , ( like 2VARIABLE )
      ;CODE   DEX, DEX,
              CLC, W LDA, 2 # ADC, BOT STA,
              TYA, W H ADC, BOT H STA, ' 2@ JMP,
      END-CODE
```

The code is very similar to that of 2VARIABLE except that the PFA is placed on the stack explicitly rather than by using PUSH. This address is replaced by the value by jumping directly into the code of 2@ which, of course, leads back to NEXT.

Incidentally, as a matter of interest, the action of ;CODE is to rewrite the code field of the most recently defined word (ie the member being defined) with the address of the byte following ;CODE. Since the compile time action of 2CONSTANT is identical to that of 2VARIABLE we could shorten the definition slightly by replacing

CREATE , ,

by

2VARIABLE

The contents of the code field will then be changed twice: once to the code of 2VARIABLE and then to the code of 2CONSTANT.

In either case entering

1234567. 2CONSTANT LONG

will create a double-precision constant LONG. Using LONG will execute the machine code following ;CODE in 2CONSTANT to place the double-precision value 1234567 on the stack.

## 10 Macro Assembly

A macro is a section of machine code which is used several times as an alternative to a subroutine. Whereas the code of a subroutine appears only once and is called whenever needed, the code of a macro is inserted into the body of the machine code at each point where it is used. The advantage of a macro is the increase in execution speed, since a subroutine call and the corresponding return are not needed. This is gained at the cost of increasing the amount of memory used, unless of course the code which is assembled is three bytes or less in length.

Creating a macro in FORTH is very similar to compiling a colon-definition. The assembly instructions are compiled and are not executed until the macro is called at a later time. The main difference is that a macro should only be created in the assembler vocabulary, as in the following examples.

ASSEMBLER DEFINITIONS HEX

```
MACRO CRLF ( ... ) ( carriage return )  
                  ( and line feed )  
  FFED JSR, ;
```

```
MACRO WRCH ( ... ) ( display accumulator )  
                  ( as ASCII char )  
  FFF4 JSR, ;
```

These two macros assemble subroutine calls, to the operating systems OSCRLF and OSWRCH respectively.

The following pair allow the assembly of a definite loop similar to the high-level DO ... LOOP. They are used in the form:

```
n TIMES, ... LOOP,
```

The loop index is stored in the Y-register which is therefore not available for other purposes within the loop body. The index is limited to the range 0 to 255 and the loop terminates when the index reaches zero.

```
MACRO TIMES, ( n ... addrB\1 )
  # LDY, BEGIN, ;
MACRO LOOP,    ( addrB\1 ... )
  DEY, 0= UNTIL, ;
```

The example below uses all four of the above macros to present the alphabet on the display. Remember that the CURRENT vocabulary must be changed back from ASSEMBLER after the definition of the macros.

FORTH DEFINITION DECIMAL

```
CODE ALPHABET ( ... )
  CRLF 64 # LDA,
  26 TIMES, CLC, 1# ADC, WRCH LOOP,
  CRLF NEXT JMP,
END-CODE
```

## 11 Errors

A comprehensive set of error checks is included with the assembler. They will not, of course, guarantee that the code definition will perform correctly but they do prevent errors of syntax.

The error checks may be divided into five main groups:

- 1 Each mnemonic when executed ensures that a valid addressing mode is used. Any illegal mode will cause error message 3 to be displayed, for example:

```
BOT )Y LDY,
```

will give

LDY, ? MSG # 3

- 2 The conditional structures include checks that they are correctly paired and nested. For example:

... IF, ... UNTIL,

will give

UNTIL, ? MSG # 19

indicating that the conditionals are incorrectly paired.

- 3 A check is also made in the conditionals that any branch is within range. For example:

... IF, 256 ALLOT THEN,

will give the error message

BRANCH TOO LONG

- 4 The number of stack items must not change accross a code definition. This can be caused, for example, by an address being left on the stack or by a conditional structure not being concluded. Such an error causes message 20 to be given.

- 5 A macro definition must be created in the ASSEMBLER vocabulary. If it is not the message

IN WRONG VOCABULARY

is given and the definition will not be accepted.

In all cases except the last the definition will be left in an incomplete SMUDGE'd state with the CONTEXT vocabulary set to ASSEMBLER.

Using SMUDGE and restoring the CONTEXT vocabulary to be the same as CURRENT will allow you to FORGET the incorrect version and try again.



## Part 2 - The Discompiler

### 1 Introduction

The discompiler is provided in eight source screens beginning at screen 36. It is loaded from FORTH by typing

```
36 LOAD
```

High-level entries in the FORTH dictionary consist of a list of the execution addresses of previously defined words. The discompiler uses these addresses to determine the appropriate names. The resulting text is similar to the source code which was originally typed.

A special feature of this discompiler is its ability to discompile headerless dictionary entries.

### 2 Using the Discompiler

There are only two words which need to be used. These are TYPE-OF and DISCOMPILE. They are used as

```
TYPE-OF      cccc
```

and

```
DISCOMPILE   cccc
```

where cccc is the name of a dictionary entry.

#### 2.1 TYPE-OF

The possible responses to TYPE-OF are

```
COLON DEFINITION  
CONSTANT  
VARIABLE  
USER VARIABLE  
VOCABULARY DEFINITION  
DOES> DEFINITION USING  cccc  
CODE DEFINITION  
UNKNOWN DEFINITION  
NOT FOUND
```

In addition, the definition will be shown as HEADERLESS and/or IMMEDIATE where

appropriate, and the execution address is given in HEX.

The values of constants and variables are also displayed in the current numeric BASE.

If a definition is of unknown type, this usually means that it is a machine code definition in which execution does not begin at the first byte of its parameter field.

## 2.2 DISCOMPILE

Any colon definition, including headerless entries, can be discompiled. If a attempt is made to discompile words other than colon definitions, e.g.

DISCOMPILE @

the message

NOT COLON DEFINITION

is given. If the word does not exist the response is:

NOT FOUND

The discompiled code will not correspond exactly with the original source code. This is because a number of words are not themselves compiled but act only as compiler directives. In some cases nothing is compiled, in others a run-time version is compiled, for example DO compiles (DO).

Most such cases are connected with either the conditionals or the literal number handling. The following examples show some sample definitions in both source and discompiled forms to illustrate the major differences. Note that all the addresses and numeric values are displayed in HEX base (with a leading #), regardless of the value of BASE at the time they were defined.

```
: TEST
    5 0 DO I . LOOP ;
```

```
DISCOMPILE TEST
LIT #5 0 (DO) I . (LOOP) #-6 ;
```

```

: CONDITION1
  IF DUP + ELSE      DUP . THEN DROP ;

DISCOMPILE CONDITION1
0BRANCH #A DUP + BRANCH #4 DUP .      DROP ;

: CONDITION2
  BEGIN DUP . 1 - DUP 0= UNTIL      ;

DISCOMPILE CONDITION2
  DUP . 1 - DUP 0= 0BRANCH #-E      ;

: PRINT
  ."  THIS IS A STRING" ;

DISCOMPILE PRINT
  (." )  THIS IS A STRING" ;

```

Additional spaces have been left in the source and discompiled code of these examples to emphasise the correspondence between the source and discompiled form.

Further examples are given below. Note that if RETURN is pressed immediately after typing TYPE-OF or DISCOMPILE the definition of 'null' (end of line) can be analysed. This is illustrated as one of the following examples:

TYPE-OF EXPECT COLON DEFINITION

CFA = #3259 OK

OK

TYPE-OF FLAG VARIABLE

VALUE 0

CFA = #8C38 OK

OK

TYPE-OF FIRST CONSTANT

VALUE -27200

CFA = #2DF6 OK

OK

TYPE-OF WIDTH USER VARIABLE

VALUE 31

CFA = #2E17 OK

OK

TYPE-OF .TYPE DOES> DEFINITION

USING CASE:

CFA = #8F0B OK

OK

TYPE-OF TYPES DOES> DEFINITION

USING LIST:

CFA = #8F23 OK

OK

TYPE-OF FORTH IMMEDIATE VOCABULARY DEFINITION

CFA = #3653 OK  
OK  
TYPE-OF + CODE DEFINITION  
CFA = #2BF2 OK  
OK  
TYPE-OF (ENTER)  
HEADERLESS CODE DEFINITION  
CFA = #3AF5 OK  
OK  
TYPE-OF (DO)  
HEADERLESS CODE DEFINITION  
CFA = #2910 OK  
OK  
TYPE-OF GARBAGE NOT FOUND  
OK  
DISCOMPILE EXPECT  
OVER + OVER (DO) KEY DUP LIT #E +ORIGIN @ =  
0BRANCH #2A  
DROP DUP I = DUP R> 2 - + >R 0BRANCH #A LIT #7  
BRANCH #6 LIT #7F BRANCH #28 DUP LIT #D =  
0BRANCH #E  
LEAVE DROP BL 0 BRANCH #4 DUP I C! 0 I 1+ !  
EMIT (LOOP) #-64 DROP ; OK  
OK  
DISCOMPILE (ENTER)  
HEADERLESS  
IN @ >R 1 BLK ! 0 IN ! INTERPRET R> IN ! 0  
BLK ! ; OK  
OK  
DISCOMPILE  
BLK @ 0BRANCH #A 0 IN ! ?EXEC R> DROP ; OK  
OK  
DISCOMPILE CASE:  
<BUILDS SMUDGE ] DOES> OVER + + @ EXECUTE ; OK  
OK  
DISCOMPILE .TYPE NOT COLON DEFINITION OK  
OK  
DISCOMPILE GARBAGE NOT FOUND  
OK